Chapter 2

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 2.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, language specification, and security-related issues. In Section 2.2, we present a summary of Software Diversification and its foundational concepts. In addition, we compare and locate our contributions against state-of-the-art related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. We finalize the chapter by stating our novel contributions.

2.1 WebAssembly overview

Over the past decades, JavaScript has been used in the majority of the browser clients to allow client-side scripting. However, due to the complexity of this language and to gain in performance, several approaches appeared, supporting different languages in the browser. For example, Java applets were introduced on web pages late in the 90's, Microsoft made an attempt with ActiveX in 1996 and Adobe added ActionScript later on 1998. All these attempts failed to persist, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Emscripten proposed with a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript itself. Asm.js was first implemented as an LLVM backend. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [40] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline or those that can be directly translated from the source code. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it

was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [31] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Despite encouraging results, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code which implies significant overhead.

Following the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially stated later by the work of Haas et al. [30] in 2017. The announcement of WebAssembly marked the first step of standarizing bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [30]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a fully Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

From source to Wasm

All WebAssembly programs are compiled ahead of time from source languages. LLVM includes Wasm as a target since version 8, supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[4]. The resulting binary, works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations. For example, if the Wasm binary is generated out of the LLVM pipeline, all optimizations in the LLVM

The step ② includes the standard library to Wasm as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4] subset of the TypeScript language

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm is final and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions.

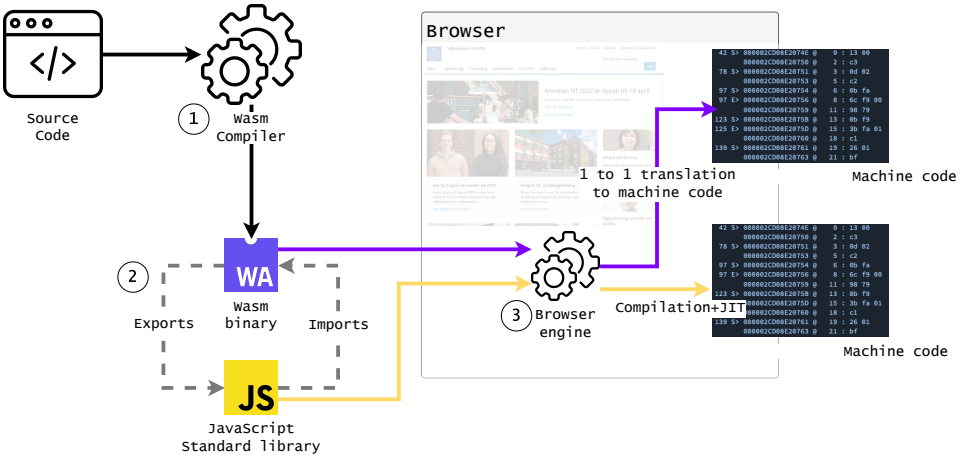

Figure 2.1: WebAssembly building, compilation in the host engine and execution.

Wasm can execute directly and is platform independent, making it useful for IoT and Edge computing [3, 16]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the bytecode alliance team [5] proposed WebAssembly System Interface (WASI). WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [18] in heterogeneous platforms.

WebAssembly specificities

WebAssembly defines its own Instruction Set Architecture (ISA) [27]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary

[5] `https://bytecodealliance.org/`

components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 2.1 and Listing 2.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm. As the listing shows, instructions are annotated with a type. Wasm binaries contain only four datatypes, `i32`, `i64`, `f32` and `f64`.

Wasm manages the memory in a restricted way.A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, however, the compiler should populate the static table of functions.

Wasm control flow structures are different from standard assembly programs. All instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.2: WebAssembly code for Listing 2.1.

```
1  (module
2    (type (;0;) (func (param f32) (result i32)))
3    (type (;1;) (func))
4    (type (;2;) (func (result i32)))
5    (import "env" "ftoi" (func $ftoi (type 0)))
6    (func $main (type 2) (result i32)
7      (local i32 i32)
8      i32.const -1000
9      local.set 0 ;loop iteration counter;
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s ;loop iteration condition;
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3 ;=12.54;
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block values. This is quarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each breaking block instruction can only jump the execution to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

We want to remark that de description of Wasm in this section follows the version 1.0 of the language and not its proposals for extended features. We follow those features implemented in the majority of the vendors according to the Wasm roadmap [28]. On the other hand we excluded instructions for datatype

conversion, table accesses and the majority of the arithmetic instructions for the sake of simplicity.

WebAssembly's security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [17] and makes easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [22]. Implementations in both browsers and standalone runtimes [3] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [25]. One of our previous contributions trigger a CVE on the code generation component of wasmtime, highlighting that even when the language specification is meant to be secure, its execution might not be. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [14] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors.

2.2 Software Diversification

The low presence of defenses implementations for WebAssembly motivates our work on Software Diversification as a preemptive technique that can help against known and yet unknown vulnerabilities. Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. We use the concept of functional equivalence defined in the seminal work of Cohen et al. [64] as input-output equivalence. Two programs are equivalent if, given identical input, they produce the identical output.

According to the survey of Baudry and Monperrus [39], the motivation for software diversification can be separated in five categories: reusability [57], software testing [50], performance [47], fault tolerance [65] and security [64]. Our work contributes to the latter two categories:

(G1) *Fault tolerance and reliability:* Mainly refers to the implementation of independent yet functionally equivalent programs for consensus during execution. Different programs are deployed and executed simultaneously; the final result is selected from all computation results. If some programs fail, the system is still able to respond. For decades, this same idea was applied to hardware. For example, in airplanes is common to have more than one sensor providing the same function. For example, Harrand et al. [15] proposed to combine the result of multiple Java decompilers.

(G2) *Security:* Mainly refers to break code reuse attacks [63] by using diverse functional programs. The main idea is to change the observable behavior of a program by changing its version every time is invoked. Thus, attackers cannot get the same information from a different source. For example, Crane et al. [38] hardened power side-channels by using diversification of software. On the same topic, Roy et al. [19] use preexisting machine learning algorithms to defeat adversarial-like attackers.

Software diversification sources.

There are two primary sources of software diversification: natural(ND) and artificial diversity(AD) [39]. Natural diversity can be controlled or an unpredicted consequence of developing processes. Controlled natural diversity is usually called Design Diversity or N-Version Diversity. It is addressed using engineering decisions [65]. In practice, it consists of providing N development teams with the exact requirements. The teams develop N independent versions using different approaches. On the other hand, Natural Diversity can emerge from spontaneous software development processes. To illustrate the Natural Diversity phenomenon, CodeForces[6] shows more than 350 different and successful solutions in C++ for a single requirements based problem in a single programming contest. The software market is an expected source of natural diversity. Sengupta et al. [29] used this fact to reach the security goal (G2).

Notice that Natural Diversity can rely on itself to escalate, and it is coped by the preexistence of software. This might be a limitation. For example, in the context o this work, the natural diversity for WebAssembly programs is nearly inexistence [6]. When natural diversity is not enough, it is innate to think that the source for diversification needs to be artificial. Automated software diversification consists of artificially synthesizing software.

According to the seminal work of Cohen et al. [64] automatic software diversification can be reached by mutation strategies. A mutation strategy is a set of rules to define how a specific piece of software should be changed to provide a different yet functionally equivalent variant. A mutation can be applied at different layers of software lifecycle, from compilation to execution and from source code to executable binary. We have found that the foundation for automatic software

[6] `https://codeforces.com/contest/1667/status/page/2?order=BY_PROGRAM_LENGTH_ASC`

diversity has barely changed since Cohen in 1993. Complemented with the work of Baudry and Monperrus [39], we enumerate the strategies for automatic software diversification.

(S1) *Equivalent arithmetic instructions* Numeric calculations can be expressed theoretically in an infinite number of ways. This strategy is simple but powerful since the complexity of program variants dramatically increases. In terms of overhead, the size of the program variant increases with the size of the replacement.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. This strategy generates program variants without affecting their size and execution time.

(S3) *Variable substitution* This strategy changes the location of variable declarations. It has a lower impact on low-level programs unless compilers resort to symbol tables. It prevents static examination and analysis of parameters and alters memory locations. The strategy should not affect the size of program variants neither their execution time.

(S4) *Adding, changing, removing jumps* This strategy creates program variants by adding, changing, or removing jumps inside the original program. At a high level, this can be reached by loop splitting or by inserting arbitrary branching. This strategy increases the execution time of variants.

(S5) *Adding, changing, removing calls* This strategy is similar to the previous one (S4). It extends the same idea by adding function calls inside the stack. It is mainly implemented by inlining and de-inlining expressions inside the program. When an expression is inlined, its original instruction is replaced by a function call that performs the same calculation as the original subexpression.

(S6) *Garbage insertion* This strategy adds instructions to the program that are independent of the original sequence of instructions. It extends S1 by supporting more instructions like random memory accesses. Dealing with code-reuse attacks, Homescu et al. [43] propose inserting garbage NOP instructions directly in LLVM IR to generate a variant with a different code layout at each compilation. Jackson et al. [46] have explored how to use NOP operations inserted during compiling time to diversify programs. Jackson et al. [46] used the optimization flags of several compilers to generate semantically equivalent binaries out of the same source code. These techniques place the compiler at the core of the diversification technique.

(S7) *Program layout randomization in memory and stack* This strategy is usually implemented on top of compilers. The generation of the final binary and how it operates its memory is randomized.

(S8) *ISA randomization* This strategy encodes the original program, for example, by using a simple XOR operation over its binary bytestream. This technique is strong against attacks involving the examination of code. It should not affect

the size of program variants or their execution times. The encoding/decoding operations are performed only once. Seminal works include instruction-set randomization [59, 61] to create a unique mapping between artificial CPU instructions and real ones.

(S9) *Simulation* It is an interpretation mechanism similar to encoding (S9), but the execution of programs is delegated to a custom interpreter instead of using preexisting execution hosts. The program is decoded at runtime every time it is invoked. Chew, and Song [62] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses), and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting.

(S10) *Intermixing* With the existence of more than one program variant, the execution of program variants can be mixed. The decision of which variant executes is decided at runtime. This strategy is the core for randomization, multivariant execution and the execution by consensus defined in G1. This strategy is complex to implement because the integrity of the memory and stack needs to be stable between programs.

Usages of Software Diversity

We categorize the applications of software diversity in three main usages.

(U1) *N-Version:* More than one independent program variant is executed in one single machine. For example, the work of Sengupta et al. [29] proposed to change indistinctly between database engines and backend server implementations in a monolithic web application. Their approach decreases the reach of exploitable CVEs from request to request. In this area, Coppens et al. [44] use compiler transformations to diversify software iteratively. Their work aims to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach continuously applies a random selection of predefined transformations using a binary diffing tool as feedback. A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them. Bathkar et al. [60, 58] proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries' memory regions, a random permutation of the order of variables and routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates an extensive range of memory error exploits. Recent work in this field includes stack layout randomization against data-oriented programming [21], and memory safety violations [4], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [11].

(U2) *Randomization:* One program that decides at runtime which of its variants to execute. Couroussé et al. [33] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. A randomization technique creates a set of unique executions for the very same program [60]. Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces required by a side-channel attack as Amarilli et al. [49] has shown. On the same topic, Agosta et al. [41] and Crane et al. [38] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize software control flow at runtime.

(U3) *Multivariant Execution(MVE):* Multiple program variants are composed in one single binary that can execute separately depending on external configuration or in parallel for consensus computation. In 2006, security researchers at the University of Virginia laid the foundations of a novel approach to security that consists in executing multiple variants of the same program, [56]. Bruschi et al. [55] and Salamat et al. [54] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on Multivariant Execution for mitigating memory vulnerabilities [23] and other specific security problems incl. return-oriented programming attacks [34] and code injection [48]. A key design decision of MVE is whether it is achieved in kernel space [20], in user-space [51], with exploiting hardware features [32], or even through code polymorphism [26]. Finally, one can neatly exploit the limit case of executing only two variants [45, 36]. Notably, Davi et al. proposed Isomeron [37], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy, a potential attacker cannot predict whether the original or the variant of a program will execute. Researching on MVE in a distributed setting like the Edge [?] has been less researched. Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [2] for the sake of efficiency.

U1, U2 and U3 can be categorized as Moving Target Defense strategies. Moving Target Defense for software was first proposed as a collection of techniques that aim to improve the security of a system by constantly moving its vulnerable components [10, 42]. Usually, MTD techniques revolve around changing system inputs and configurations to reduce attack surfaces. This increases uncertainty for attackers and makes their attacks more difficult. Ultimately, potential attackers cannot hit what they cannot see. MTD can be implemented in different ways, including via dynamic runtime platforms [19].

Statement of Novelty

We contribute to Software Diversification for WebAssembly using Artificial Diversification, for N-Version, Randomization and Multivariant Execution usages (U1, U2, U3) for the sake of reliability and security (G1 and G2). The primary motivation for our contributions is that we see in WebAssembly a monoculture problem. If one environment is vulnerable, all the others are vulnerable in the same manner as the same WebAssembly binary is replicated. Besides, the WebAssembly environment lacks natural diversity [39]. Compared to the work of Harrand et al. [?] , in WebAssembly, one could not use preexisting and different program versions to provide diversification. The current limitations on security and the lack of preexisting diversity motivate our work on software diversification as one preemptive mitigation among a wide range of security countermeasures.

In Table 2.1 we listed related work on Software Diversification that support our work. The table is composed by the authors and the reference to their work, the source of diversification (natural or artificial), followed by one column for each motivation, strategy and usage (G1, G2, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, U1, U2 and U3). Each cell in the table contains a checkmark if the strategy, the motivation or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

As the table illustrates, we push state of the art in Software Diversification. Our first contribution, CROW [9] generates multiple program variants for WebAssembly using the LLVM pipeline. It contributes to state of the art in artificially creating diversity for WebAssembly. While the number of related work for software diversity is large, only one approach has been applied to the context of WebAssembly. To the best of our knowledge, the closest diversification work on the browsers involving WebAssembly is the work of Romano et al. [1]. They proposed to de-inline JavaScript (S5) subexpressions and replace them with function calls to WebAssembly counterparts. The presence of two different engines, one for JS and another for WebAssembly in the majority of the browser vendors, motivated their work. They empirically demonstrated that malware classifiers could be evaded with this diversification technique. On the other hand, WebAssembly is a novel technology, and the adoption of defenses for it is still under development [3, 5].

CROW, extrapolates the idea of superdiversification [53] for WebAssembly. CROW works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures, something not possible with the x86-specific approach of previous works. CROW focuses on the static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a randomization approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. It also can be used in fuzzing campaigns [?] to provide reliability. The diversification created by CROW can

Authors	ND	AD	G1	G2	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	U1	U2	U3
Chew and Song [62]		✓							✓							✓	
Kc et al. [59]		✓		✓								✓	✓				
Barrantes et al. [61]		✓		✓								✓				✓	
Bhatkar et al. [60] and Bhatkar et al. [58]		✓		✓		✓	✓			✓	✓					✓	
Cox et al. [56]	✓			✓										✓			✓
Bruschi et al. [55]		✓		✓							✓			✓		✓	
Salamat et al. [54]		✓		✓					✓					✓			✓
Jacob et al. [53]		✓			✓		✓			✓					✓		
Salamat et al. [51] and Österlund et al. [20]		✓									✓						✓
Amarilli et al. [49]				✓	✓								✓			✓	
Jackson [46] and Homescu et al. [43]		✓								✓							
Coppens et al. [44]	✓																
Agosta et al. [41]	✓			✓													✓
Crane et al. [38]		✓		✓		✓			✓	✓				✓			✓
Davi et al. [37]		✓		✓										✓		✓	
Couroussé et al. [33]				✓		✓				✓			✓	✓		✓	
Koning et al. [32]		✓		✓							✓						✓
Sengupta et al. [29] and Roi et al. [19]	✓			✓										✓			✓
Lu et al. [23]		✓		✓							✓			✓			✓
Belleville et al. [26]		✓		✓	✓					✓	✓					✓	
Aga et al. [21] and Lee et al. [4]		✓		✓							✓					✓	
Xu et al. [11]		✓		✓							✓					✓	
Harrand et al. [15]	✓													✓			
Voulimeneas et al. [2]		✓												✓			✓
Cabrera Arteaga et al. [9]		✓		✓	✓	✓	✓	✓		✓					✓	✓	
Cabrera Arteaga et al. [8]		✓		✓	✓	✓	✓	✓	✓	✓				✓		✓	✓

Table 2.1: Comparison table of related work. The table is composed by the authors and the reference to their work, the source of diversification (natural or artificial), followed by one column for each motivation, strategy and usage (G1, G2, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, U1, U2 and U3). Each cell in the table contains a checkmark if the strategy, the motivation or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order.

unleash hidden behaviors in compilers and interpreters. Thanks to CROW, a bug was discovered in the Lucet compiler [7]

Moreover, with MEWE [8], we contribute to the field of Software Diversification at two stages. First, we automatically generate variants of a given program with CROW [9]. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable behaviors at each run of the program. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE.

[7] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-p`

Conclusions

In this chapter, we presented the background on the WebAssembly language, including our motivation for its security issues and related work enforcing its security. Software Diversification has been widely researched, not being the case for WebAssembly. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly. We stated our contributions to the field of artificial diversity by extending the current state of the art. Our contributions are obtained by following the methodology described in Chapter 3.